Fast Walsh–Hadamard Transform (FWHT) Executive Report

The Hadamard matrix and Walsh functions

$$H_1 = [1] \qquad H_2 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$

$$H_4 = H_2 \otimes H_2 = \begin{bmatrix} H_2 & H_2 \\ H_2 & -H_2 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \end{bmatrix}$$

- The **Hadamard matrix** H□ is an n×n orthogonal matrix (up to scale) with entries only **+1 or −1**.
- Its **rows** are the **Walsh functions**: sequences of ±1 values arranged according to a specific order (e.g., natural, sequency, dyadic).
- Each Walsh function row can be constructed as the **product of simpler square waves** (Eg. Row 4 of H_4 is row 2 by row 3 element-wise).
- **Sequency:** the number of sign changes in a Walsh function row.
    - Low sequency rows change sign rarely (like low frequency).
    - High sequency rows change sign often (like high frequency).

Thus, applying the Walsh–Hadamard Transform corresponds to multiplying a vector x by a Hadamard matrix H□

What you *can do* with FWHT (capabilities / use cases)

1. **Global mixing / one-to-all connectivity**

    - A single input value (or a single nonzero row/column) spreads to *all* outputs after FWHT stages. This makes the transform a cheap global mixer — ideal for fast diffusion of information, embeddings mixing, or expanding a narrow channel into a wide mixed representation.

2. **Sparse → structured dense conversion**

    - Multiplying or composing a sparse matrix with Hadamard transforms (left and/or right) converts a sparse pattern into a structured dense matrix whose structure is determined by Hadamard basis signs. Useful to implement dense-like mixing while storing only sparse factors (memory and compute savings).

3. **Low-cost orthogonal transforms**

    - Add/sub operations only (no multiplies except optional normalization). Efficient for integer data and low-precision implementations.

4. **Locality-sensitive hashing & random projections**

    - WHT combined with simple sign/quantization yields fast LSH and Johnson–Lindenstrauss style embeddings with low cost and fixed, deterministic pseudo-random mixing.

5. **Feature spreading / preconditioning for learning**

    - Replace or augment dense linear layers with WHT mixing layers to increase receptive field and reduce parameter count.

6. **Fast XOR-convolution (subset / bitwise convolutions)**

    - For problems where convolution index uses XOR (e.g., subset DP / convolution), FWHT implements convolution via transform → pointwise multiply → inverse transform.

7. **Compression-friendly transforms**

    - Basis is ±1 square waves (sequency ordering useful for energy compaction in some signals). Works for simple transform coding and quantization.

8. **Hashing & fingerprinting**
    - Deterministic, inexpensive mixing makes FWHT useful for sketching and hashing pipelines.
9. **Combining multiple narrow layers into a wide one**
    - Use the one-to-all property to take several small-width blocks, then WHT-mix them — implementing a wide effective (neural network) layer cheaply.
10. **Fast associative operations / correlation**
    - Enables certain associative lookups and similarity measures with predictable latency and hardware friendliness.
11. **Streaming, in-place transforms for memory-limited systems**
    - In-place FWHT needs no extra buffer beyond the input, good for streaming and embedded applications.

Mathematical applications & properties

Core properties (short)

1. **Orthogonal (up to scale):** $H^T = H$ and $H \cdot H = n \cdot I$. So inverse is $(1/n) \cdot H$.
2. **Basis of ±1 square waves:** rows are sequences (simple products of square waves) of ±1 where sequency measures sign changes.
3. **Self-inverse:** applying FWHT twice yields a scalar multiple of the identity (recover original up to factor).
4. **O(n log n) arithmetic:** only additions/subtractions in the fast algorithm (no multiplies except for normalization).
5. **Transforms XOR convolution to pointwise multiply (for XOR convolution problems).**

Mathematical/applicational list

1. **Data compression / transform coding**
    - Energy compaction in some signals; cheap transform for pre-quantization. Sequency ordering can help allocate coefficients.
2. **Random projections / dimensionality reduction**
    - Use randomized sign flips + WHT (the “SRHT-like” pattern) to implement Johnson–Lindenstrauss embeddings fast (often combined with subsampling and scaling).
3. **Locality-sensitive hashing (LSH) and sketching**
    - Deterministic mixing + simple quantization provides low-cost sketches.
4. **Sparse→dense matrix conversion & structured dense design**
    - By left/right multiplying sparse factors with Hadamard matrices you can emulate dense transforms with compact representation. For example:
        - D is sparse diagonal or block-sparse; compute H·D·H (or permutations) to produce a dense, structured mixing matrix cheaply.
    - This is the “one-to-all” exploitation: a few nonzeros in D get spread across outputs.
5. **Neural-network architectural uses**
    - Replace expensive fully-connected layers or global mixing with H-structured mixing (cheap, parameter-free).
    - Use to merge small parallel layers into one wide output via WHT mixing and small learned diagonal/projection matrices.
6. **XOR / subset convolution and combinatorial algorithms**
    - FWHT is the established fast method for XOR convolution in algorithmic problems (subset DP).
7. **Preconditioning & whitening**

    - As an orthogonal transform, WHT can be used as a preprocessing step to decorrelate components or as a cheap approximate whitening.

8. **Fast correlation / similarity measures**
    - After transform, certain correlations and inner products can be computed efficiently and used for fast similarity search.

9. **Error-resilient and deterministic mixing**
    - Deterministic, repeatable mixing is useful in systems where randomness is undesirable or constrained.

10. **Hardware-friendly codes & multiplexing**
    - WHT basis can be used for lightweight CDMA-like multiplexing in constrained radios or sensor networks.

In-place FWHT Algorithm

Modifies the input array directly, no extra buffer needed.

Idea:

- Start with blocks of size 2. For each lower, upper *half block* pair, compute the **sum** and **difference**. Place:
    - sum in the **lower half** of the block,
    - difference in the **upper half**.

Then repeat with block sizes doubling each stage: 4, 8, 16, …, until size = n.

Implementation (Java)

```
public class WHTSimple {
// In place fast Walsh Hadamard transform (FWHT)
    public static void wht(float[] vec) {
        final int n=vec.length;  // n must be a power of 2 like 2,4,8...
        // Break the vector into blocks of size hs*2
        int hs=1; // starting with a block size of 2
        while (hs < n) {                // Until the block size becomes too large
            int i = 0;                  // Current index position in vec array
            while (i < n) {             // Until that index becomes too large
                int j = i + hs;         // j=the index + half a block
                while (i < j) {         // While i is in the lower half of the block
                    float a = vec[i];   // Get an element in the lower half of the block
                    float b = vec[i + hs];          // And the element from the corresponding upper half
                    vec[i] = a + b;     // Put the sum into the lower half
                    vec[i + hs] = a - b;// Put the difference in the upper half of the block
                    i += 1;             // Next index in the block to process
                }
                i += hs;                // Move i to the next block to process
            }
            hs += hs;                           // Vector has been fully processed at current
        }                                       // block size, double block size and repeat
    }

    public static void main(String[] args){
                        float[] x={1f,2f,3f,4f,5f,6f,7f,8f};
                        wht(x);
                        System.out.println(java.util.Arrays.toString(x));
                        wht(x);
                        System.out.println(java.util.Arrays.toString(x));
                }

}
```

Booklet:

[FWHT Booklet]